情報処理学会論文誌　データベース　Vol.10　No.1　24–39 (Mar. 2017)

並列ログ先行書き込み手法P-WAL

神谷 孝明[1,a)]　川島 英之[2,b)]　星野 喬[3,c)]　建部 修見[2,d)]

受付日 2016年9月10日, 採録日 2017年1月8日

概要：本研究はフラッシュストレージをログ用のストレージデバイスとするときにふさわしい WAL プロトコルとして P-WAL を提案する．フラッシュストレージは複数のメモリチップに対して並列にアクセスすることで高い性能を発揮する．P-WAL はフラッシュストレージの特性を活用し，各ワーカが専用の領域にログを書き込む並列ログ書き込み方式を用いる．この方式により従来の直列 WAL 方式で発生する，排他制御処理とストレージ I/O にともなう性能低下問題を解決する．P-WAL をトランザクションシステム上で実装し，性能評価を行った．その結果，P-WAL は直列 WAL 方式に対してマイクロベンチマークで 10.0 倍，TPC-C ベンチマークにおいて 2.3 倍の性能向上を示した．

キーワード：ログ先行書き込み，WAL，トランザクション，フラッシュストレージ

P-WAL: Paralell Write-ahead Logging

Komei Kamiya[1,a)]　Hideyuki Kawashima[2,b)]　Takashi Hoshino[3,c)]　Osamu Tatebe[2,d)]

Received: September 10, 2016, Accepted: January 8, 2017

Abstract: This paper proposes a new WAL protocol, P-WAL. We first demonstrate that parallel write operations well perform on a flash storage. P-WAL exploits the features of the flash storage. P-WAL lets each worker writes log records to its dedicated storage space. This design eliminates both the contentions on WAL buffer and the inefficient I/O operations where the conventional sequential WAL method suffers from. We design and implement P-WAL on a prototype transaction manager, and evaluate it with benchmarks. The result of experiments showed that P-WAL outperformed the conventional WAL. The improvement factors were 10.0 on micro-benchmark and 2.3 on TPC-C benchmark respectively.

Keywords: write-ahead logging, WAL, transaction, flash storage

1. はじめに

1.1 WAL：ログ先行書き込み

現代の情報システムの中核をなすトランザクション処理システムにおけるボトルネックの 1 つはログ先行書き込み（WAL）[18] である．WAL はシステム障害に備えてデータの更新前にログを書き込む方式であり，WAL によりトランザクションの原子性と永続性が保証される．従来の WAL アルゴリズムはストレージデバイスとして HDD を前提としており，HDD の高い I/O コストによる性能劣化を最小限にするため，複数のトランザクションログをメモリ中で直列化し，ストレージデバイスに追記する方式をとる．この方式を**直列 WAL** と表記する．直列 WAL の概要図を**図 1** に示す．図では複数のワーカスレッドが 1 つの WAL バッファにログレコードを挿入し，WAL バッファ内のすべてのログレコードが一括して HDD へ書き込まれる様子が示されている．

直列 WAL には 2 つの性能ボトルネックが存在する．第 1 の性能ボトルネックは［**B1**］**WAL バッファアクセス**

1　筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
2　筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
3　サイボウズ・ラボ株式会社
Cybozu Labs, Inc., Chuo, Tokyo 103–6028, Japan
a)　kamiya@hpcs.cs.tsukuba.ac.jp
b)　kawasima@cs.tsukuba.ac.jp
c)　hoshino@labs.cybozu.co.jp
d)　tatebe@cs.tsukuba.ac.jp

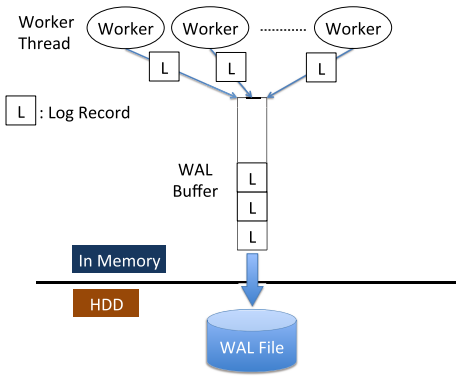

図 1　直列 WAL のアーキテクチャ
Fig. 1　Architecture of serial WAL.

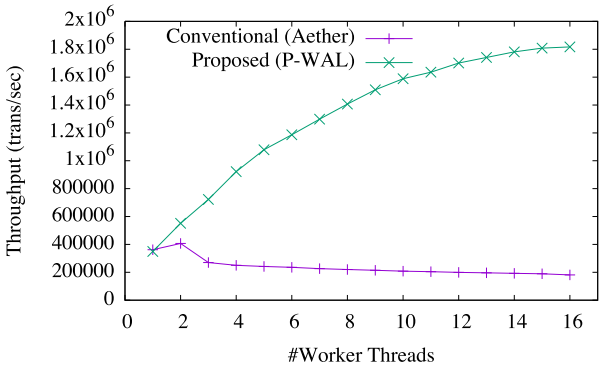

図 2　従来手法（直列 WAL）と提案手法（並列 WAL）の比較
Fig. 2　A comparison of the conventional method (serial WAL) and the proposed method (parallel WAL).

の競合である．複数のワーカスレッドがただ 1 つの WAL バッファを共有するため，WAL バッファへのログレコード挿入には排他制御を要する．すなわち，あるワーカスレッドが WAL バッファへ挿入処理をしている間，他のワーカスレッドはその終了を待機する必要がある．

第 2 の性能ボトルネックは **［B2］WAL バッファの永続化待機**である．WAL バッファに含まれるログは，当該トランザクションのコミットが確定しクライアントに結果を返す前にストレージ上ですべて永続化されていなくてはならない．ログの永続化は，複数のトランザクションのログをまとめて書き込むグループコミット [5], [21] および，表側のバッファを永続化中に裏側のバッファにログの挿入を許すダブルバッファリング [22] により効率化される．しかし，ダブルバッファリングを用いたとしても，バッファの永続化操作は直列化される必要があるため，表側のバッファの永続化中に裏側のバッファを永続化開始することはできず，表側のバッファの永続化完了を待機する必要がある．

1.2　フラッシュストレージ

直列 WAL はストレージデバイスが HDD であることを前提に設計されている．一方で，近年は SSD を中心とするフラッシュストレージが従来の HDD の代替として広く普及が進んでいる．フラッシュストレージの大きな特徴は，HDD と比較してレイテンシが小さいことと，ランダムアクセスによる性能低下が少ないことである．また，フラッシュストレージは一般に複数のメモリチップ（チャネル）が搭載されており，これらに対して並列にアクセスすることで高い性能を発揮する．フラッシュストレージ上でワーカスレッドの数を変えながら，直列 WAL の従来手法 Aether [8] のトランザクション処理性能を測定した結果を**図 2** の Conventional (Aether) に示す[*1]．この結果より，直列 WAL はマルチコア環境下においてフラッシュストレージの性能を十分に引き出すことが難しいと示唆される．

[*1] このデータは図 7 からの抜粋である．詳細は 5.3 節で述べられる．

1.3　研究課題

本論文の研究課題は，フラッシュストレージをログ用のストレージデバイスに用いた場合に，WAL の性能を最大化することである．フラッシュストレージの特性は前述のとおり，レイテンシが小さく，高い並列アクセス性能を持ち，ランダムライトによる性能低下が少ない点である．したがってその性能を最大限に引き出すには，**WAL の並列化**が求められる．WAL の並列化を達成するために，複数の WAL バッファを用いて，WAL バッファを並列に書き込む手法が考えられる．しかしながら，障害発生後はトランザクションログを用いて障害前までに記録された操作を時系列順に再適用し，データベースを障害直前の状態にリカバリできなければならないため，複数の WAL ファイルを用意して並列に書くだけではリカバリ可能とならない．また，WAL バッファの数が増えても競合が発生すれば性能は低下するため，［B1］を解決するためには競合を極小化する設計が求められる．そして［B2］を解決すべく各 WAL バッファを並列に書き込むと，ログ欠損という事態が生じる可能性がある．グループコミットのように複数トランザクションのログを同時に書き込むことにより，ログ欠損の可能性はいっそう増える．そのような状況下でも，リカバリが失敗しないプロトコル設計が求められる．

1.4　貢献

本論文の貢献は以下の 3 つである．

- 既存研究の passive group commit [26] はバイト単位のアクセスが可能な NVM を対象に WAL の並列化手法を提案したが，本論文はすでに広く使われているブロック単位でアクセスするフラッシュストレージを対象に WAL 並列化手法 P-WAL を提案したこと．
- フラッシュストレージは複数チャネルを持つため，並列アクセスによってストレージの持つ性能をより引き出せる．本論文では評価実験を行い，P-WAL の並列アクセスが直列 WAL と比較してより高いスループットとスケーラビリティを達成し，それを裏付けるデー

タを示したこと．
- 既存研究の Silo [25], [28]/FOEDUS [9] はトランザクション遅延が大きくなるがスループットをさらに向上させる手法である epoch 方式を WAL 技術において提案したが，P-WAL は epoch 方式ほどトランザクション遅延を犠牲にせずに，直列 WAL よりもスループットを向上させることを実験により示したこと．

1.5　本論文の構成

本論文の構成は以下のとおりである．2 章では各ストレージデバイスの特性を述べる．3 章では高性能並列 WAL として P-WAL を提案し，その通常時のプロトコルを述べる．4 章では P-WAL リカバリプロトコルを述べる．5 章では P-WAL の評価を述べる．6 章では関連研究を述べる．7 章では結論を述べる．

2.　ストレージデバイスの性能特性

直列 WAL はストレージデバイスが HDD であることを前提に設計され，ログを単一の WAL ファイルに追記する方式を用いている．一方，フラッシュストレージは性能特性が HDD とは異なるため，フラッシュストレージをログ記録に用いた場合，直列 WAL 方式が適しているかは不明である．本章では，フラッシュストレージを前提とした WAL プロトコルを解説する準備として，その書き込み特性を調査した結果を述べる．

2.1　フラッシュストレージの概要

近年，SSD や USB メモリなどのフラッシュストレージが HDD の代替として広く普及が進んでいる．フラッシュストレージは一般的に，NAND 型フラッシュメモリとコントローラからなる．フラッシュストレージの特徴として，HDD と比較してレイテンシが小さいことと，ランダムアクセスによる性能劣化が少ないことがあげられる．これはフラッシュストレージが HDD とは異なり，I/O の際に機械的な動作を必要としないためである．また，フラッシュストレージは一般に複数のメモリチップ（チャネル）が搭載されており，これらに対して並列にアクセスすることで高い性能を発揮する．

一方で，フラッシュメモリの書き換え操作は，データを一度消去してから再度データを書き直す必要がある．この消去操作にかかる時間はデータの読み書きよりも著しく大きいことが知られている [14]．また，読み書きの単位のページサイズ（典型的に 512 Bytes または 4 KiB）よりも消去の単位のブロックサイズは大きい（典型的に 16 KiB または 128 KiB）ことから，同一消去ブロック内の書き換えの必要がないページまで一緒に消去され，残すべきページも書き直しをともなう．この消去操作を繰り返すことでフラッシュメモリの寿命が短くなることから，フラッシュストレージでは FTL（Flash Translation Layer）を用いて，フラッシュメモリ上の論理空間と物理空間をマッピングテーブル上で管理し，特定のページに書き込みが集中しないように，ウェアレベリングを行うことが一般的である．

インタフェースとしては SATA や SAS などのほかに，近年の CPU はそのコントローラを内蔵している PCI Express に接続することで，より小さいアクセス遅延で，より大きい帯域を使用する製品も存在する．フラッシュストレージは通常のブロックデバイスとしてアクセスすることが可能な一方，フラッシュストレージ用の特別なインタフェースを提供する製品も存在する．

このインタフェースにアクセスするための API として，OpenNVM [20]，Flood [2] などがストレージベンダから提供されている．本論文の実験で用いる ioDrive [3] は OpenNVM をサポートしている．OpenNVM を用いることで OS カーネルをバイパスしてデバイスへ直接書き込みができるほか，API の 1 つである nvm_batch_atomic_operations を使用すると，複数ブロックへの書き込みや trim をアトミックに実行できる．各ブロックへの操作がすべて成功したときに一連の操作が有効となり，1 つでもブロックの操作に失敗するとすべての操作が無効になる．これらの API を用いてデータベースを高速化する取り組みも行われている [16]．

現時点では前述の API を利用できるのはそれらをサポートする特定の製品に限られるが，近年フラッシュストレージや NVM 向けの標準規格である NVMe [19] が策定された．このことから，将来的には通常の OS のデバイスドライバが NVMe をサポートし，これらのデバイスに適したインタフェースを介してアクセスすることが一般的になると予測される．

2.2　各種デバイスの基本性能評価

並列ログ書き込みに用いたときのデバイスの性能特性を理解するために，WAL 書き込みを模擬[*2]して，事前に割り当てられた互いに重複しない連続領域に対して，各スレッドが同期書き込みを繰り返すプログラムを用いて評価する．O_SYNC と O_DIRECT を指定して open したデバイスに対し，write システムコールを用いて同じ IO サイズで同期的にシーケンシャル書き込みを行い，HDD，SSD，ioDrive のスループットを測定した．パラメータとして，スレッド数，IO サイズを変化させ，それぞれのパラメータで 10 回測定し，平均値をプロットした．

HDD と ioDrive の測定環境を**表 1**，SSD の測定環境を**表 2** に示す．HDD と SSD は RAID コントローラに接続されているため，RAID コントローラを介してアクセスす

[*2] このワークロードでは WAL 書き込みを模擬するため同期処理が多数生じる．したがってこの実験結果はシーケンシャルライトよりも低性能である．

表 1　ioDrive と HDD の測定環境
Table 1　Measurement environment (w/ ioDrive and HDD).

CPU	Intel Xeon E5-2665 (8cores)× 2
Memory	64 GB
OS	CentOS release 6.8 (Linux kernel 2.6.32)
ioDrive	Dell ioDrive (Model: VRG5T) PCI Express 2.0 x8, SLC, 160 GB
RAID Controller	PERC H710P, PCI-Express 2.0 x8
HDD	Dell (Model: MBE2147RC) 2.5 inch SAS 2 6 Gb/s, 15,000 rpm

表 2　SSD の測定環境
Table 2　Measurement environment (w/ SSD).

CPU	Intel Xeon E5620 (4cores) × 2
Memory	24 GB
OS	CentOS release 6.8 (Linux kernel 2.6.32)
RAID Controller	PERC H710P, PCI-Express 2.0 x8
SSD	Patriot Wildfire (Model: PW120GS25SSDR) 2.5 inch SATA 3.0, MLC, 120 GB

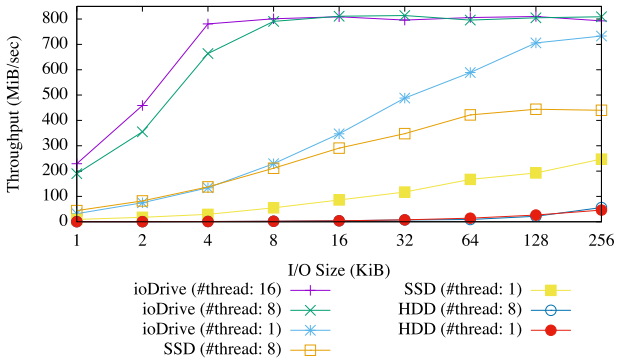

図 3　各種デバイスの書き込み性能評価
Fig. 3　Write performance on different devices.

る必要がある．本実験では RAID コントローラの HDD と SSD への書き込みポリシを write through にすることで，RAID コントローラのキャッシュを使用しないようにしている．

実験結果を**図 3** に示す．SSD が搭載されたマシンは HDD・ioDrive が搭載されたマシンとは異なる環境であるが，実験中の CPU 使用率が 100%に達しておらず，PCI Express 2.0 x8 の帯域は片方向 4 GB/sec であり，メモリ容量も十分大きいことから，ストレージ I/O が律速要因になっていることが示唆されるため，同一グラフ上に示した．

書き込み I/O サイズの増大にともない，いずれもスループットが向上している．SSD や ioDrive では複数スレッドで 1 つのデバイスに書き込んだときに，ランダムライトが引き起こされると考えられるが，複数のメモリチップに並列にアクセスすることにより，複数スレッド時の性能は 1 スレッド時のそれを上回る．また，並列度が高いと，より小さな I/O サイズでピーク性能に達することが分かる．これより，フラッシュストレージでは並列度の高さが高性能化を導くことが示唆される．

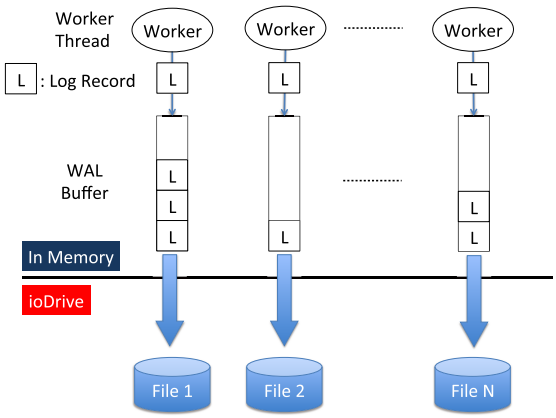

図 4　P-WAL のアーキテクチャ
Fig. 4　Architecture of P-WAL.

2.3　直列 WAL の問題点

直列 WAL において性能低下の要因となるのが，**［B1］ログレコードの WAL バッファ挿入時の競合**と**［B2］WAL バッファの永続化待機**である．WAL バッファ挿入時の競合は，Aether で提案されている scalable log buffer により緩和されるが，それによる性能向上は最大 7%[*3]にとどまる．複数のワーカが単一の WAL バッファを共有する設計では，競合にともなう性能低下を避けることが困難だと考えられる．

また，従来の HDD などのディスクストレージでは，スループットを最大化するためにランダムアクセスが発生しないようログを単一の WAL ファイルに追記する必要がある．しかし，フラッシュストレージにおいては，1 つのデバイスに対して 1 スレッドで追記書き込みを行うよりも，複数のスレッドで並列に書き込むことでさらなるスループット性能の向上が達成される（図 3）．

3.　P-WAL：並列 WAL

3.1　通常処理のプロトコル

近年のマルチコア環境上でフラッシュストレージを活用し，直列 WAL の問題点を解決すべく，並列 WAL プロトコル P-WAL を提案する．**図 4** に P-WAL のアーキテクチャを示す．P-WAL は各ワーカスレッドに専用の WAL バッファを割り当てる．したがって WAL バッファアクセスに際して，ワーカスレッド間の競合は生じず，競合回避の機構は不要になる．これにより［B1］は解決される．

また，P-WAL ではワーカスレッドごとに専用の WAL

[*3]　詳細は文献 [8] の 6.4 節と図 9 を参照されたい．

ファイルを持たせる．この設計によりP-WALはWALファイルの書き込みに際して，他のワーカスレッドのI/O完了の待機が不要になる．これにより［B2］は解決される．

この設計により，P-WALはWALバッファへのログレコード挿入とWALファイルへの書き込みを自然な形で並列化する．

3.1.1　WALバッファへのログレコード挿入

各WALバッファの操作は対応するただ1つのワーカスレッドのみが行う．そのためログレコード挿入（WALバッファの領域確保とログレコードのコピー）に排他制御を必要としない．一方，複数のWALバッファを用いることで，ログレコードの順序関係が不明になる可能性がある．

直列WALでは，単一のWALバッファにログレコードを追記していくため，末尾に近いものほど新しいログであと判別可能である．P-WALにおいても各WALバッファ内ではこれが成立するが，異なるWALバッファに存在するログレコード間では順序関係が定まらず，そのままではクラッシュリカバリが不可能である．

そこでP-WALではログレコードの一意な順序番号であるLSN（Log Sequence Number）[4]を共有カウンタを用いて割り当てる．直列WALでは，LSNは永続化されたログレコードのファイル上の位置を指し示す役割も兼ねる．一方P-WALにおいて共有カウンタを用いて発行されるLSNは，ファイル上の位置とは無関係である．そのためワーカスレッドはLSN以外に格納先のWALファイル番号とファイル内のオフセットをLSNの付属情報として別途記録する．P-WALでは，共有カウンタへのアクセスがWALバッファへのログレコード挿入時の唯一のクリティカルセクションとなる．共有カウンタの値の読み込みとインクリメントにはロックを使用せず，アトミック命令である（fetch_and_add）を用いることで，クリティカルセクションを短縮する[*4]．

また，後述するコミット条件の判定を単純化するため，同一トランザクションのログは同一WALバッファに挿入する．すなわち，トランザクションは1つのワーカスレッドに割り当てられる．

3.1.2　ログレコードの永続化

WALバッファ内のログレコードはまとめてWALファイルに永続化される．直列WALでは，単一のWALバッファを用いるため，WALバッファに先頭から順にログレコードを挿入し，WALバッファを先頭から永続化していけば，ログレコードはLSN順に永続化される．

一方P-WALでは，WALバッファが複数存在するため，ログレコードがLSN順に永続化される保証はない．システム障害時に，WALファイル上に永続化されているログレコードのLSNに欠損が発生する可能性がある．P-WALではこの問題に対処するために，自分自身とそれより小さいLSNを持つログレコードがすべて永続化されている場合にのみそのログを有効とする．

P-WALにおけるトランザクションのコミット条件は，該当トランザクションのコミットログが有効であることとする．あるトランザクションのすべてのログレコードは同一WALバッファを使用し，コミットログ以下のLSNを持つ．また，各トランザクションがコミットログのLSNを取得後にロックを解放することを前提とすれば，あるトランザクションのコミットログのLSNを取得するとき，そのトランザクションがアクセスしたリソースに以前にアクセスしたトランザクションのすべてのログのLSNは，当該コミットログのLSNよりも小さい．すなわち，コミットログが有効であるならば，これらのログもすべて有効であり，コミット条件が満たされる．

3.1.3　クライアント通知の制御

P-WALにおけるトランザクションのコミット条件はコミットログとそれ以下のLSNを持つすべてのログが欠損なく永続化されていることである．この条件を満たす前にクライアントにトランザクションの結果を返してしまうと，障害が起こった場合，アボート扱いになるべきところをコミットされたと間違った結果をクライアントに返したことになる可能性が生じる．そのため，トランザクションがコミット条件を満たすまで，クライアントへの処理完了の通知は保留される．

各ワーカスレッドはコミットログをWALバッファに挿入後，トランザクションID（XID）とコミットログのLSN（commitLSN）の情報を持つエントリをローカルのコミットキューに追加する．また，各ワーカスレッドはWALバッファの永続化が完了すると，自身が最後に永続化を完了したLSN（flushedLSN）の情報を更新する．すべてのワーカのflushedLSNの最小値が，欠損なくログレコードが永続された最大のLSNと見なせるので，各ワーカは定期的に，すべてのワーカのflushedLSNの最小値（min(flushedLSNs)）と，コミットキューのエントリのcommitLSNを比較し，commitLSN <= min(flushedLSN)を満たすトランザクションがあった場合，キューから取り出して，クライアントにトランザクション処理完了を通知する．

障害によってLSNに欠損が発生した場合，欠損が発生したLSNよりも大きいLSNのログレコードを持つトランザクションは，障害時点でコミットが完了していない．そのため，クライアントにリプライは返しておらず，したがってそのままアボートされる．

3.2　Aetherの適用

Aether [8]はearly lock release（ELR），flush pipelining（FP），そしてscalable log buffer（SLB）と呼ばれる高性

[*4] 我々の測定環境ではfetch_and_addがCASおよびmutexよりも高速であった．

能化手法を提案している．これらのうち，ELR と FP を P-WAL に適用する手法を本節で述べる．なお，P-WAL が SLB を用いない理由は，それが不要だからである．SLB はログバッファへの挿入処理の競合問題を緩和する技術である．P-WAL ではログバッファの孤立化により，その競合問題を完全に解決するため，SLB は不要になる．

3.2.1　Early Lock Release（ELR）の適用

ELR は，トランザクションログのフラッシュが完了する前に，獲得しているリソースのロックを解放する技術である．ELR を行う場合は，リソースの更新順序を反映させるために，リソースへのロックを獲得した順にトランザクションをコミットしなければならない．直列 WAL では，WAL バッファ上でログが直列化されている．そのため，WAL バッファにコミットログを挿入した後にロックを解放することで，その後に同じリソースへのロックを獲得するトランザクションのログは後から追記される．したがって直列 WAL では単純に WAL バッファの先頭からログを永続化するだけで，この条件が満たされる．

一方 P-WAL ではロック獲得順序とトランザクションコミット順序の対応付けが必要である．そこで P-WAL では，先で述べたようにコミットログとそれ以下の LSN を持つすべてのログが永続化されていることを，トランザクションのコミット条件とする．コミットログの LSN を取得した後にロックを開放するため，ロック獲得順序が先である方がコミットログの LSN が小さくなる．これにより，コミット条件が満たされる順序は，ロック獲得順序と同一となる．そのため，P-WAL においても直列 WAL 同様，ELR を適用し，ロックを獲得した順にトランザクションをコミットすることが可能である．

3.2.2　Flush Pipelining（FP）の適用

グループコミットは，実際に I/O を行うスレッドは 1 つであり，他のワーカスレッドはその I/O の完了を待たなければならない．これは，トランザクションが結果をクライアントに返すためには，少なくともログが永続化されている必要があるからである．Flush Pipelining（FP）は，WAL バッファの永続化を待たずに，ワーカスレッドが新たなトランザクションの処理を開始する方式である．ワーカスレッドの代わりに WAL バッファの永続化を行うデーモンスレッドを用意し，グループコミットと組み合わせることで，複数のトランザクションログをまとめて書き込む．WAL バッファの永続化が完了すると，デーモンスレッドはワーカスレッドに通知を行い，ワーカスレッドはコミットされた担当のトランザクションの結果をクライアントに返す．

P-WAL で FP を用いる方式を提案する．P-WAL には WAL バッファが複数存在するため，WAL バッファの永続化を行うデーモンスレッドは用意しない．各ワーカスレッドは WAL バッファにコミットログを書いた後，I/O を発行する代わりに，新たなトランザクションの処理を開始する．そして指定されたサイズの複数のトランザクションログが溜まるのを待ってから，ワーカスレッド自身が WAL バッファの永続化を行う．

Algorithm 1 ワーカへのトランザクションの割当て

```
 1: function run
 2:     ntx ← 0
 3:     while runnable() do
 4:         tx ←attachTx()
 5:         executeTransaction(tx)
 6:         ntx ← ntx + 1
 7:         if ntx = N then
 8:             self.flushedLSN ←flushLog()
 9:             ntx ← 0
10:         end if
11:         controlNotification()
12:     end while
13: end function
```

Algorithm 2 トランザクションの実行

```
 1: function executeTransaction(tx)
 2:     lockQueue ←<>                          ▷ <> is empty queue
 3:     beginTx()
 4:     for all op ∈ tx.ops do
 5:         locks ← executeOperation(op)
 6:         lockQueue.push(locks)
 7:     end for
 8:     tx.commitLSN ← endTx()
 9:     self.commitQueue.push(tx)
10:     releaseLocks(lockQueue)
11: end function
```

3.3　並行実行制御を含む通常処理

並行実行制御と WAL の両方を含むトランザクションの通常処理の流れを，擬似コードの形式で Algorithm 1〜4 に示す．ワーカスレッドはシステムの起動時に固定数作成され[*5]，各々は run 関数（Algorithm 1）を実行する．runnable 関数（Algorithm 1 - line 3）は，通常処理の受け付け時に true を返す．attachTx 関数（Algorithm 1 - line 4）は，ワーカスレッドにトランザクションを割り当てる．受け取ったトランザクションの情報をもとに，executeTransaction 関数（Algorithm 1 - line 5）が呼び出される．

トランザクションを実行する executeTransaction 関数を Algorithm 2 に示す．トランザクションは開始時は，どのリソースに対してもロックを獲得していない（Algorithm 2 - line 2）．beginTx 関数（Algorithm 2 - line 3）は BEGIN ログを作成するなどのトランザクションの開始にともなう処理を行う．トランザクションは複数の操作（選択/更新/挿入/削除）から構成される．各操作について executeOperation 関数を呼び出す（Algorithm 2 - line 5）．

[*5] 図 3 より，ストレージのスループットを向上させるためには，CPU のコア数を超えないなるべく多くのワーカスレッドを作成することが良いと考えられる．

Algorithm 3 オペレーションの実行

```
 1: function executeOperation(op)
 2:     locks ←<>                          ▷ <> is empty queue
 3:     for all rid ∈ op.rids do
 4:         lock ← acquireLock(rid)
 5:         locks.push(lock)
 6:     end for
 7:     insertLogRecord(op)
 8:     operate(op)
 9:     return locks
10: end function
```

操作を実行する executeOperation 関数を Algorithm 3 に示す．rid は操作対象とするリソースの番号を指し示している（Algorithm 3 - line 3）．1 つの操作でロックの獲得対象のリソースは複数存在する可能性があるため，それらのすべてのリソースに対して必要な shared/exclusive ロックを獲得する（Algorithm 3 - line 3〜6）．このプロトコル上では省略するがデッドロックは適切に回避，または検知されロールバックされるものとする．insertLogRecord 関数（Algorithm 3 - line 7）は，操作内容を記述したログレコードを WAL バッファに挿入する．operate 関数（Algorithm 3 - line 8）は，リソースに対する操作を実行する．

各操作で獲得されたロックはまとめてキューに入れて管理される（Algorithm 2 - line 5〜6）．endTx 関数（Algorithm 2 - line 8）はコミットログ（END ログ）を作成し，そのコミットログの LSN を返す．コミットログの LSN を含んだトランザクションの情報は，コミット待ちのキューに入れられる（Algorithm 2 - line 9）．コミットログを WAL バッファに入れた段階で，作成したトランザクションログを永続化する前に，トランザクション中に獲得したすべてのロックを解放する（Algorithm 2 - line 10）．すなわちロッキングプロトコルは 2 phase locking を用い，ロック解放タイミングは ELR と通知制御（3.1.3 項）により早期化される．

一般に一度に書き込むサイズが大きいほど，ストレージのスループット性能は高くなる傾向にあるので，N 件のトランザクションのログが溜まったとき（Algorithm 1 - line 7）に，WAL バッファのログを WAL ファイルにまとめて書き込み，永続化する（Algorithm 1 - line 8）．この直後に，ワーカの flushedLSN の値を最後に書き込んだログレコードの LSN に更新する．

トランザクションのコミット以外のすべて処理が完了したら，通知制御（3.1.3 項）によりコミットできるトランザクションがないか調査する（Algorithm 1 - line 11）．controlNotification（Algorithm 4）では初めに全ワーカの flushedLSN の最小値を求める（Algorithm 4 - line 3〜8）．各ワーカは自らが管理しているコミット待ちのキューを先頭からチェックし（Algorithm 4 - line 10），コミットログの LSN が flushedLSN の最小値以下のトランザクションがあれば（Algorithm 4 - line 11），そのトランザクションはコミットされたと見なし，クライアントに結果を返す（Algorithm 4 - line 12）．以上の処理をワーカは繰り返す．

Algorithm 4 クライアントへの返送制御

```
 1: function controlNotification
 2:                ▷ TYPE_MAX means available maximum number
 3:     min ← TYPE_MAX
 4:     for all worker ∈ workers do
 5:         if min > worker.flushedLSN then
 6:             min ← worker.flushedLSN
 7:         end if
 8:     end for
 9:     while self.commitQueue.notEmpty() do
10:         tx ← self.commitQueue.front()
11:         if min ≥ tx.commitLSN then
12:             reply(tx)
13:             self.commitQueue.pop()
14:         else
15:             break
16:         end if
17:     end while
18: end function
```

4. P-WAL のリカバリ

P-WAL は WAL ファイルを分割する．これにより，従来の ARIES のリカバリプロトコルが使用不能になる．ナイーブな解決策は全ファイル内の有効なログレコードの整列だが，その数を N としたとき，O（NlogN）の高いコストを要する．そこでマージ方式を提案する．

4.1 各ログの順序の決定

従来の直列 WAL 方式では共通の WAL バッファにログレコードを挿入し，それをそのまま WAL ファイルに追記する．そのため，リカバリ時には WAL ファイルの先頭からログを読んでいけば，時系列順にログを処理できる．一方，P-WAL 方式では，ワーカスレッドの数だけ WAL ファイルが生成される．各 WAL ファイル内のログレコードの順序関係は，末尾に近い方が新しいものだと判別可能だが，異なる WAL ファイル上にあるログレコード間の順序関係は不明である．そこで単調増加するログの ID である LSN をもとに，WAL ファイル間でのログの順序関係を解決する．リカバリの際は，LSN が小さい順にログを処理する．

4.2 マージ方式

各 WAL ファイルにおいて，ログレコードは時系列順，すなわち LSN 順に記録されている．この特徴に基づき，各 WAL ファイルの先頭ログレコードの LSN を見て，LSN の値が最も小さいログレコードから順に処理する．

図 5 にマージ処理の例を示す．まず，各 WAL ファイルの先頭ログレコード（LSN = 1, 2, 5）の中で，最も LSN の小さい LSN = 1 のログレコードを処理する．File 1 の

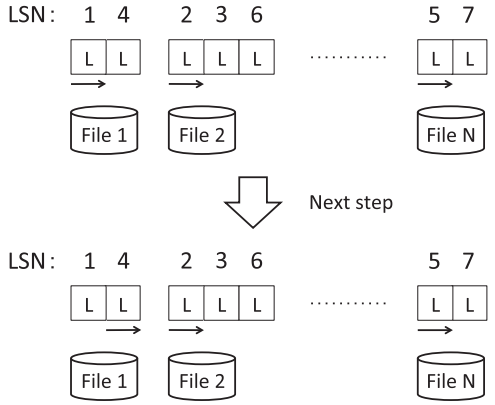

図 5　マージ処理によるリカバリ
Fig. 5　Merge recovery.

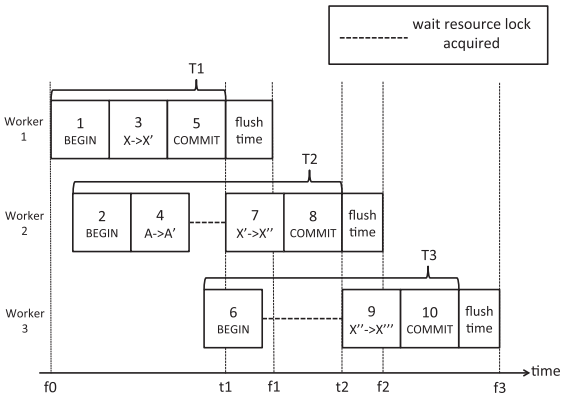

図 6　トランザクションログの生成時系列の例
Fig. 6　Example of time series for generating transaction log records.

ポインタを先に進めると，LSN = 4 のログレコードが File 1 の次の先頭のログレコードとなる．その次は，各 WAL ファイルの先頭ログレコード（LSN = 4, 2, 5）の中で，最も LSN の小さい LSN = 2 のログレコードを処理する．このように各 WAL ファイルの先頭のログレコードの LSN を比較し，LSN の小さい順にログを処理することで障害発生前までの歴史を繰り返すことができる．

4.3　障害時の LSN 欠損への対処

P-WAL では障害時に WAL ファイルの各ログレコードに記載された LSN に欠損が発生する可能性がある．しかし，その状態から整合性が保たれた状態にリカバリが可能である．

ログレコードには数種類あり，トランザクション開始時は BEGIN ログ*6，リソースの更新時は UPDATE ログ，トランザクションのコミット時は COMMIT ログを生成する．あるトランザクションがコミットされたと見なせるのは，そのトランザクションのコミットログに記載された LSN 以下の LSN の全ログが永続化されたときである．リカバリ時，このコミット条件を満たすトランザクションによる更新のみを復元する．通知制御（3.1.3 項）を適用しているため，ここでアボート扱いになったトランザクションはすべてクライアントに未通知のものである．

例として，同じリソース X を操作する，すなわち依存関係のあるトランザクション T1，T2，T3 について，トランザクションログを時系列で示したものを**図 6** に示す．T1，T2，T3 は異なるワーカスレッドによって並列に処理され，P-WAL の設計の下，ログもそれぞれ異なる WAL バッファに溜められるものとする．四角は 1 つのログレコードを指し，ログレコード内の数字は LSN を表す．LSN の下に，ログレコードの内容を記載している．X→X' の表記は，リソース（ページやレコード）X を X から X' に更新する UPDATE ログを意味する．同様に，X'→X'' の表記は，リソース X を X' から X'' に更新する UPDATE ログを意味する．

T1 はトランザクション実行中に X に対する write ロックおよびその他の読み込み対象のリソースに対する read ロックを獲得する．T1 が獲得した write ロックは ELR により，T1 のコミットログを WAL バッファに挿入した（$t1$）後に行われる．T1 が開放したリソース X の write ロックを次に T2 が獲得し，同様に T2 がコミットログを WAL バッファに挿入した（t2）後にリソース A，X の write ロックを開放し，次に T3 がリソース X の write ロックを獲得する．

トランザクション T1，T2，T3 のログはそれぞれ $f1$，$f2$，$f3$ の時点で初めて永続化が完了する．$f0 \leq t < f1$ の間に障害が発生した場合，ログは 1 つも永続化されておらず，コミットされたトランザクションは存在しない．$f1 \leq t < f2$ の間に障害が発生した場合，T1 のトランザクションログが永続化されている．このとき，T1 のコミットログの LSN：5 以下の LSN：2，4 のログレコードは永続化されていない．そのため，T1 はコミットされていないものと見なす．$f2 \leq t < f3$ の間に障害が発生した場合，T1 と T2 のトランザクションログが永続化されている．このとき，T1 のコミットログの LSN：5 以下の LSN の全ログが永続化されているため，T1 はコミットされたものと見なす．一方で，T2 のコミットログの LSN：8 以下の LSN：6 のログレコードが永続化されていないため，T2 はコミットされていないものと見なす．$f3 \leq t$ 間に障害が発生した場合は，T1，T2，T3 のすべてのトランザクションログが永続化されている．T1 は $f2$ の時点でコミット条件を満たしており，T2，T3 に関してもそれぞれコミットログの LSN 以下の LSN の全ログが永続化されているため，T1，T2，T3 はコミットされたものと見なす．

その後は ARIES のプロトコルに従い，コミットされたトランザクションによる更新を復元し，コミットされなかったトランザクションによる更新はなかったものとする．

*6　BEGIN ログは必ずしも必要ではない [18].

以上より，WAL ファイル上のログに LSN の欠損が発生していてもトランザクションのコミット条件をチェックすることで，整合性のとれた状態へのリカバリが可能であることが示された．

グループコミットのように複数のトランザクションログを同時に書き込む場合も，各トランザクションのコミットログの LSN を見て，その LSN 以下の全ログが永続化されている場合のみ，そのトランザクションはコミットされたと見なすことで，同様にリカバリが可能である．

5. 評価実験

5.1 実験条件

本章では P-WAL の性能を実験的に評価する．以降すべての実験は表 1 の環境で ioDrive を用いて行った．

実験では 4 つの手法 Aether，Aether++，P-WAL，P-WAL++を評価する．Aether は Aether [8] で提案された手法であり，flush pipelining[*7]，early lock release，scalable log buffer を含む．Aether++は Aether に対して，write システムコールの代わりに，API（OpenNVM batch_atomic_operation）を用いてログの書き込みを行う手法である．P-WAL は 3 章，4 章で述べた提案をすべて取り入れた手法である．P-WAL++は Aether++と同様に，P-WAL に対して write システムコールの代わりに，同 API を用いてログの書き込みを行う手法である．

これらの各手法について，ファイルシステムを介さずにデバイスファイル上に WAL 領域を設定し，で複数のトランザクション（特に断りのない場合は 16 個）のログを同時に書き込む[*8]．デバイスファイルは Linux では “/dev/” 以下に存在する．デバイスファイルへの書き込みは，ファイルシステムへの書き込みとは異なり，どこからどこまでを 1 つのファイルの範囲とするかを，自システムで決めたルールに従って管理しなければならない．ファイルシステムを介した書き込みに比べて，デバイスファイルへの直接書き込みはファイルシステムをバイパスできるためデバイスの性能上限に近い I/O 性能が期待できる．ログを書き込む際には一度に書き込む単位であるチャンクごとにチェックサム（CRC-32 Castagnoli [10]）を計算し，その値をチャンクのヘッダに記録している．これにより，ログを読み込む際には初めにチャンクのヘッダを読み，その後に続くデータが永続化に成功したか失敗したかをチェックサムを用いて判断することが可能である．

[*7] 文献 [8] は flush pipelining に専用のコミットデーモンスレッドを用いるが，本実験で実装した flush pipelining は 1 つのワーカスレッドをコミットスレッドとして用いる．コミットスレッド以外のワーカスレッドはトランザクションがコミットする前に，次のトランザクションの処理を開始できるという点は Aether と同じである．

[*8] グループコミットと同様の操作だが，P-WAL においてはその永続化完了のみでコミット条件が満たされるわけではないことに注意．

5.2 ワークロード

5.2.1 マイクロベンチマーク

本項ではマイクロベンチマークを用いて P-WAL の基本性能を評価する．このマイクロベンチマークには 2 種類のワークロード, U-1 と U-10 がある．U-1 における各トランザクションは更新処理を 1 回実行する．そのログは [BEGIN, UPDATE, END] となる．U-10 は U-1 の負荷を高めたワークロードであり，更新処理を 10 回含むトランザクションを実行する．そのログは [BEGIN,UPDATE×10,END] となる．

ログレコードは可変長でログの種類によってサイズが変わる．BEGIN ログ，END ログは 132 バイト，UPDATE ログは 156 バイトである．WAL バッファは WAL ファイルに書き込まれる際，512 バイト単位の 1 つのチャンクとして書き込まれ，空き領域はパディングされる．

各オペレーションは，メモリ上にある総数 65,536 のページの中からランダムに 1 つのページを選択して操作を適用する．各ページには 32 bit 整数オブジェクト 1 つのみが含まれており，オペレーションに応じて読み込み操作あるいは更新操作を行う．ロックはページを対象として行われる．

5.2.2 TPC-C

この実験では，TPC-C ベンチマークの New-Order トランザクションを用いる．New-Order トランザクションは，商品の注文処理を模擬したワークロードである．

1 回の注文で購入する商品の種類の数（ol_cnt）は，ベンチマークアプリケーションによってランダムに生成される．なお，本実験で 1 つのトランザクションが生成するログサイズの平均は 4,414 バイトである．パラメータの warehouse 数は 8 に設定した．トランザクションのロールバックは発生させず，通常処理を繰り返したときの性能を計測する．

5.3 スループット測定結果

トランザクションとストレージのスループットに関する実験結果を U-1 については**図 7**，**図 8**，U-10 については**図 9**，**図 10**，TPC-C について**図 11**，**図 12** に示す．グラフの各点は 10 回の測定の平均値である．

5.3.1 U-1

図 7 から次のことが分かる．(R1) スレッド数の増加にともなう性能向上が P-WAL++と P-WAL では 16 スレッドまで観察される一方，Aether++と Aether では 2 スレッドまでしか観察されない．Aether においては，スレッドの数を増やすことにより，WAL バッファのアクセス競合頻度が増加し性能低下が起こったと考えられる．一方で，P-WAL は WAL バッファのアクセス競合を緩和し，ストレージへの並列書き込みを行うため，スレッド数を増やすことにより性能が 16 スレッドまで向上したと考えられる．(R2) P-WAL の最高性能は 16 スレッド時の 1.82 M tps である．

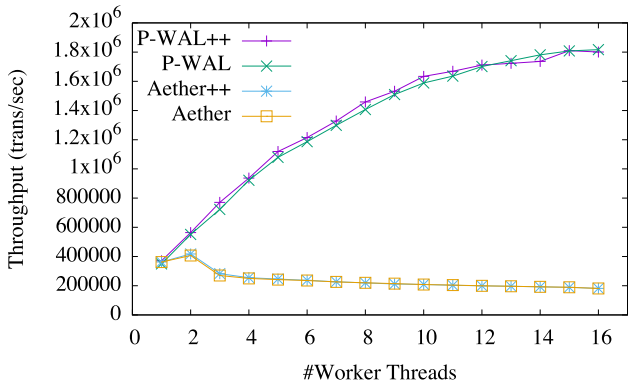

図 7　U-1：トランザクションのスループット
Fig. 7　U-1: Transaction throughput.

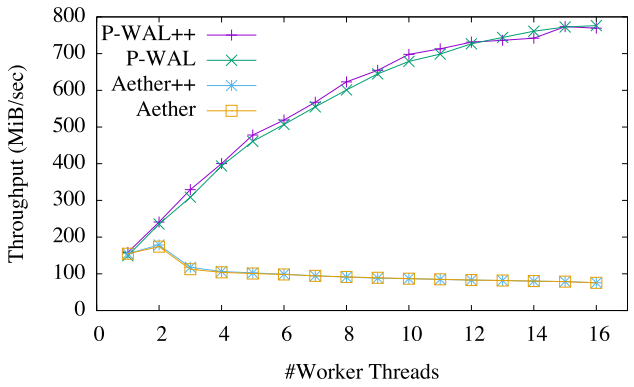

図 8　U-1：ストレージのスループット
Fig. 8　U-1: Storage throughput.

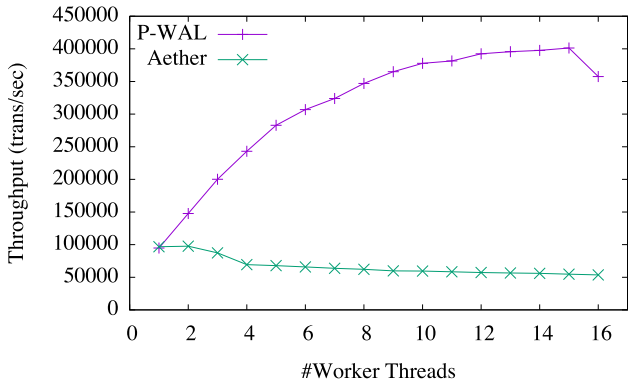

図 9　U-10：トランザクションのスループット
Fig. 9　U-10: Transaction throughput.

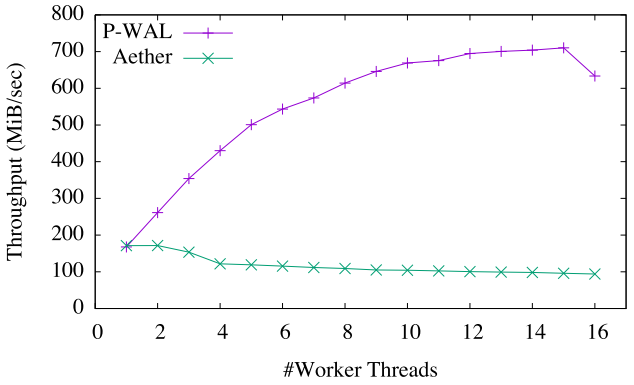

図 10　U-10：ストレージのスループット
Fig. 10　U-10: Storage throughput.

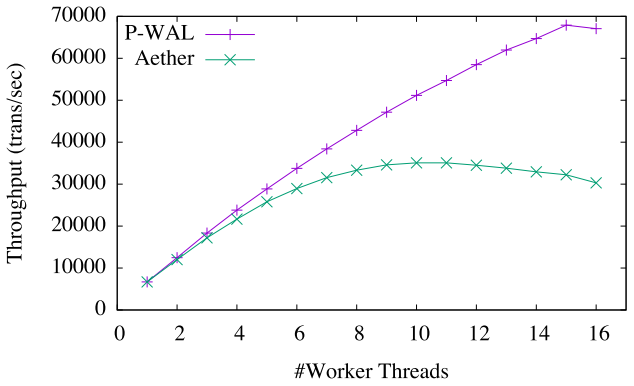

図 11　TPC-C：トランザクションのスループット
Fig. 11　TPC-C: Transaction throughput.

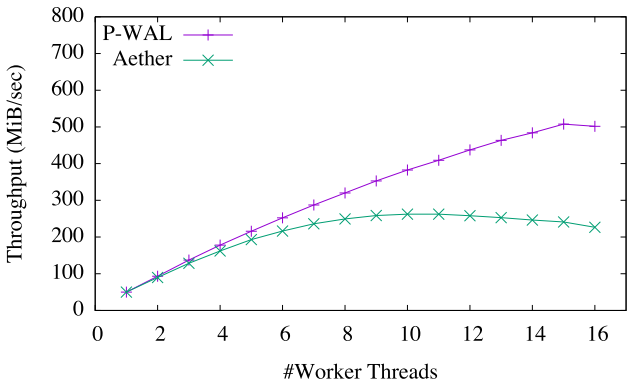

図 12　TPC-C：ストレージのスループット
Fig. 12　TPC-C: Storage throughput.

(R3) P-WAL++と P-WAL の性能差の平均は 2.0%でほとんど差がない．P-WAL は API を使わないものの，ファイルシステムを介さずに DIRECT I/O でログを書いているために OS レイヤのオーバヘッドは少なく，API を用いた P-WAL++による性能差がほとんど見られなかったと考えられる．(R4) P-WAL と Aether の最大性能差は 16 スレッド時の 10.0 倍である．

図 8 から次のことが分かる．(R5) P-WAL の最高ストレージスループットは 16 スレッド時の 776 MiB/sec である．P-WAL が，U-1 で一度に書き込むログのチャンクサイズは 7168 バイトである．このサイズに近い 8 KiB の I/O サイズ，16 スレッドで P-WAL のログ書き込みを模擬した際のデバイスの性能（図 3 参照）は 801 MiB/sec である．776 MiB/sec はこの値の 97%にあたり，P-WAL はストレージデバイスの性能を十分に活用することができているといえる．

(R3) より，OpenNVM の利用は性能向上をもたらすものの，本実装における効果は小さい．以降の図は P-WAL，Aether のみを記載する．

5.3.2　U-10

図 9 と図 10 から次のことが分かる．(R6) スレッド数の増加にともなう性能向上が P-WAL は 15 スレッドまで観察される一方，Aether では 2 スレッドまでしか観察されない．これは，U-1 と同様に Aether においては WAL バッファアクセスの競合で性能が飽和したと考えられる．(R7) P-WAL の最高性能は 15 スレッド時の 0.40 M tps であり，その値は Aether の 7.3 倍である．(R8) P-WAL において，16 スレッド時に大きな性能低下が見られた．CPU の物理コア数は 16 であり，U-10 は U-1 と比べてトランザクション実行時間の中で CPU 時間の割合が大きく，本プログラムのマスタスレッドをはじめ，OS 上で他のプログラムなども同時に動いていることから，ワーカスレッドに割り当てる CPU のコア数が足りなくなったと考えられる．(R9) P-WAL の最高ストレージスループットは 15 スレッド時の 710 MiB/sec である．P-WAL が，U-10 で一度に書き込むログのチャンクサイズは 28.5 KiB バイトである．このサイズに近い 32 KiB の I/O サイズ，16 スレッドで P-WAL のログ書き込みを模擬した性能（図 3 参照）は 796 MiB/sec である．710 MiB/sec この値の 89%にあたり，U-1 ほどではないが，U-10 においても P-WAL はストレージデバイスの性能を十分に活用することができているといえる．

5.3.3　TPC-C

図 11 と図 12 から次のことが分かる．(R10) スレッド数の増加にともなう性能向上が P-WAL は 16 スレッドまで観察される一方で，Aether では 10 スレッドまでしか観察されない．Aether では U-1, U-10 では性能向上が 2 スレッドまでしか観察されないが，TPC-C では 10 スレッドまで観察された．この理由は，TPC-C は U-1，U-10 と比べて，ログの書き込みが必要なデータの更新・挿入操作以外にも多くの参照操作が含まれており，WAL バッファの競合が緩和されているため，スレッド数の増加にともない，並列にトランザクションを処理したことにより性能が向上したものと思われる．(R11) P-WAL の最高性能は 16 スレッド時の 67.8 K tps であり，その値は Aether の 2.3 倍である．(R12) P-WAL の使用ストレージ帯域の最大値は 15 スレッド時の 508 MiB/sec である．P-WAL が，TPC-C で一度に書き込むログのチャンクサイズは平均で約 69 KiB バイトである．このサイズに近い 64 KiB の I/O サイズ，16 スレッドで P-WAL のログ書き込みを模擬した際のデバイスの性能（図 3 参照）は 806 MiB/sec である．508 MiB/sec はこの値の 64%にあたり，TPC-C においては，P-WAL はストレージデバイスの性能を十分に活用することができていないといえる．これは，U-1，U-10 と比べて TPC-C はトランザクション実行時間の中でリソースのロック待ちを含む CPU 時間の割合が大きく，I/O を十分な頻度で発行できなかったためであると考えられる．

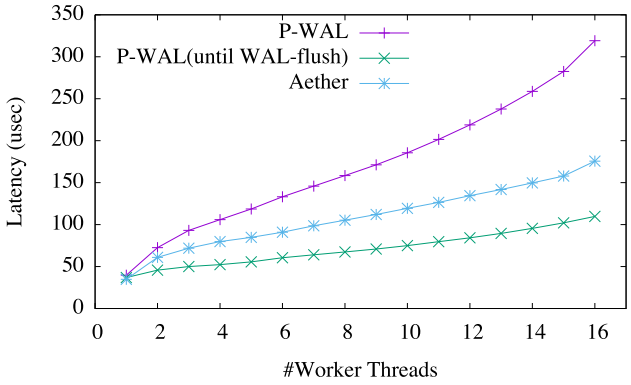

図 13　U-1：遅延
Fig. 13　U-1: Latency.

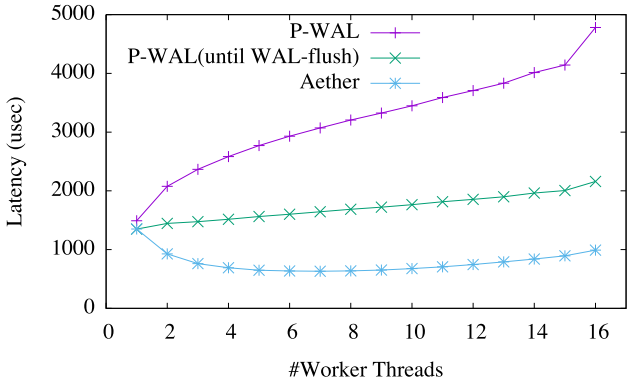

図 14　TPC-C：遅延
Fig. 14　TPC-C: Latency.

5.3.4　スループット測定実験のまとめ

以上の結果は次のようにまとめられる．

(1) (R1, R2, R4, R6, R7, R10, R11) より，P-WAL は高いスケーラビリティがある．
(2) (R5, R9) より，P-WAL は U-1，U-10 では ioDrive の帯域を十分に活用できている．
(3) (R12) より，P-WAL は TPC-C では ioDrive の帯域を活用しきれていない．

5.4　遅延の評価

次に遅延という観点から P-WAL を評価する．データベースシステムなどの大量のタスクを並列に処理するシステムにおいて，スループットを向上させるソフトウェア技術は遅延を犠牲にすることが多く [9], [25], [28]，P-WAL もまた例外ではない．5.1 節に示した条件で，U-1 と TPC-C において遅延を測定した．トランザクションにおける遅延は，あるトランザクションの開始時刻からコミットが確定しその結果がクライアントへ通知可能になる時刻（コミット判明時刻）までの差である．

グループコミットサイズを 16 トランザクションとしたときの，測定結果を図 13 と図 14 に示す．なお，P-WAL においては，複数のトランザクションログを同時に書き込むだけでコミット条件が満たされるわけではない．P-WAL

ではトランザクションの遅延に加え，トランザクションの開始時刻からそのトランザクションのすべてのログが書き込まれた時刻までの差（until WAL-flush）を示す．Aetherにおいては，あるトランザクションについて，そのトランザクションのすべてのログが永続化された時刻とトランザクションのコミット判明時刻は等しいが，P-WALにおいては，自分のトランザクションのログを含め，それ以前のログがすべて永続化された状態になったことが確認された時刻がコミット判明時刻となる．

(R13) P-WALは，遅延の中でログを書き終わってからコミットが判明するまでの時間（コミット待機時間）の比率が，スレッド数の増加にともない大きくなる．遅延におけるコミット待機時間の比率は，U-1では1スレッドのときに12%，4スレッドのときに25%，16スレッドのときに44%である．TPC-Cでは1スレッドのときに10%，4スレッドのときに41%，16スレッドのときに55%である．これは，スレッド数の増加にともない，より多くの数のWALバッファの永続化を待たなくてはならなくなったためだと考えられる．

(R14) P-WALは，U-1，TPC-Cともにスレッド数が増加するとWAL-flushまでに要する遅延が増加する．スレッド数が増えるほどI/Oの発行頻度が増えるため，あるI/O要求を発行してから，その要求を受け付けて完了するまでの遅延が増加した可能性が考えられる．

(R15) Aetherは，U-1ではスレッド数の増加にともない遅延が増加する一方，TPC-Cでは7スレッド目まではスレッド数の増加にともない遅延が減少し，以降は遅延が増加する．Aetherは，共有のWALバッファを用いるため，多数のスレッドからのアクセスによりCPUキャッシュの無効化が発生しやすいため，スレッド数の増加にともない遅延が増加したと考えられる．TPC-CはU-1と比べて，ログの書き込みが必要なデータの更新・挿入操作以外にも多くの参照操作が含まれているため，WALバッファの競合が緩和され，スレッド数の増加にともない，WALバッファ内でグループコミットのサイズのトランザクションログが溜まるまでの時間が短くなったと考えられる．

(R16) P-WALはU-1，TPC-Cともにスレッド数の増加にともないコミット判明時刻までの遅延が増加する．P-WALは16スレッド時に最大の遅延を示し，U-1では330マイクロ秒程度，TPC-Cでは4.7ミリ秒程度である．16スレッド時のAetherの遅延性能はU-1についてはAetherの1.8倍，TPC-Cについては4.8倍である．Aetherよりも P-WALは遅延が大きいため，低遅延性を重視するアプリケーションにおいてP-WALが有利とはいえない．

(R17) WAL-flushまでの遅延はU-1においてはP-WALの方が小さく，TPC-Cにおいては Aetherの方が小さい．この理由として，U-1においてはP-WALはAetherよりも早くトランザクションログをグループコミットのサイズまで溜められた可能性がある．なぜならWALバッファの競合を緩和した設計のP-WALはキャッシュの効率的な利用が可能だからである．

近年の高スループットOLTPシステム（例：Silo [25], [28]，FOEDUS [9]）では，遅延はepochなるパラメータに左右される．epochは通常40ミリ秒であり，WAL操作を除いたトランザクションの実行時間がepochに対して十分短かったとしても，実際の遅延は平均20ミリ秒以上になることが予想される．一方，本実験で確認したP-WALの最大遅延はTPC-Cにおける16スレッドのときの4.7ミリ秒であり，P-WALはSilo [25], [28] やFOEDUS [9] と比較して，遅延の犠牲を比較的避けながらスループットを向上させる技法に位置づけられる．

5.5　書き込み性能低下の調査

フラッシュストレージでは書き込みにより，フラッシュストレージの論理・物理アドレスマッピング書き換えと，そのガベージコレクションなどの負荷が発生する．そのため，書き込みの継続が性能低下を誘発する．この現象はagingとして一般的に知られている．また，フラッシュストレージにおける物理的な上書きは，16 KiB以上の消去ブロック単位で実行される．そのため，数KB単位のランダム書き込みによってwrite amplificationは発生する．

これらagingとwrite amplificationの影響を調査するため，P-WALを模擬して，事前に割り当てられた互いに重複しない連続領域に対して16ワーカスレッドがそれぞれ長期書き込みを実施する．また，データの物理的な上書きによる消去操作が発生する前に，消去可能な論理領域をあらかじめフラッシュストレージに通知するtrim操作を行うことで，不要な論理・物理アドレスマッピングのエントリの削除や，データの物理消去を効率的に行える可能性がある．そのため，trimを行わない場合と行う場合の両方について性能経過を調査する．本実験では，ioctlにBLKDISCARDを指定することによってdiscard I/Oを発行することでtrim操作を行った．

ログ領域に対してtrimを行う際は，その領域のログが不要になっていなければならない．ストレージ上には存在せず揮発メモリ上にのみ存在するデータ（ダーティデータ）を，ストレージに書き出すことで，そのデータを更新したログは不要になる．この処理はチェックポイントと呼ばれる．直列WALの場合，チェックポイントによるログのtrimはWALファイルを2つに領域分割すれば実現できる．たとえば，各WALファイルを4 GiBとすると，1つ目のWALファイルをすべて使いきった後は，以降のログは2つ目のWALファイルに追記するようにし，2つ目のWALファイルを使いきる前に，1つ目のWALファイルに書いたログに対応する更新操作をチェックポイントですべてストレージ上に永続化する．チェックポイント完了

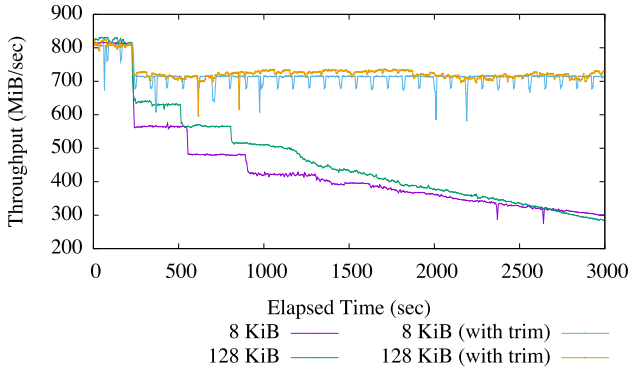

図 15　長期書き込みとストレージスループットの関係
Fig. 15　Storage throughput with long-term write workloads.

後は 1 つ目の WAL ファイルは不要になるため trim を行うことが可能になる．2 つの WAL ファイルのローテートを繰り返すことで継続的にログの書き込みと trim を行うことが可能である．P-WAL においては，各ワーカスレッドごとに，WAL ファイルを 2 つ用意し，WAL ファイルのローテーションを同期することでこれに対応可能である．

パラメータとして書き込み I/O サイズには 8 KiB と 128 KiB を用いた．書き込みを行うワーカスレッド数は 16 とした．ワーカスレッドは 256 MiB を書き込むたびに，平均スループットを更新し，マスタスレッドは 5 秒おきに，各ワーカスレッドの平均スループットを合計した全体のスループット性能を計測する．Trim を行う場合は，各ワーカスレッドが 4 GiB 書き込むたびに，直前に書いていた 4 GiB の領域を trim する．

長期書き込みを実施した結果を**図 15** に示す．これより次のことが分かる．

(R18) いずれのブロックサイズでも徐々に性能が低下する．そのため，aging 現象が発生していることが示唆される．(R19) 128 KiB よりも 8 KiB の方が，性能低下が急峻である．そのため，write amplification が発生していることが示唆される．一方，5.3 節で述べた実験中には上記の現象は発生しなかった．その理由としてはワークロード中に I/O 以外の処理が含まれていたこと，実験での書き込み量が ioDrive の容量である 160 GB 以内だったことなどが理由だと考えられる．(R20) Trim を行うことで，いずれのブロックサイズにおいても，長期書き込みにおける性能低下を防ぐことができている．P-WAL はチェックポイントと trim を組み合わせることで，一定の性能を保ちながら継続的にトランザクション処理を行うことが可能であると示唆される．(R21) Trim を行う場合では，定期的に 40〜100 (MiB/sec) 程度の性能低下が発生している．これは定期的に消去操作が発生し，一時的な書き込み性能の低下が起こったと考えられる．

6. 関連研究

トランザクションの原子性と永続性を担保しながらトランザクション処理を高性能化するために，ストレージデバイスの特性に合わせた WAL に関する研究が多く行われてきた．

6.1 HDD を前提とする研究

ARIES [18] は WAL とリソースへの細粒度ロックを用いたトランザクションのリカバリプロトコルを提案した．広く使われているデータベースシステム（MySQL，PostgreSQL など）をはじめ，HDD を前提とする研究の多くが ARIES のスキームに基づいている．

一方で，ARIES のスキームにおいては，WAL やリソースアクセスに要する排他制御がトランザクション処理のボトルネック要因となっていた [6]．これらの問題は，CPU のコア数が多くなるにつれより深刻になることが予測されるため，Aether [8] はリソースへのロック解放タイミングの早期化である early lock release（ELR），グループコミット待ち処理の非同期化である flush pipelining（FP），そして WAL バッファへのログ挿入処理の競合緩和である scalable log buffer（SLB）を組み合わせ，WAL を高速化できることを示した．Aether では，WAL バッファと WAL ファイルが 1 つだけ存在するが，本研究は，WAL バッファと WAL ファイルがワーカスレッドの数だけ存在する．

D-ARIES [23] は，分散環境向けに ARIES を拡張し，各ノードがそれぞれローカルにログを書き込む方式を提案した．並列にログを書き込むという点は本研究と類似するが，本研究は単一ノードのマルチコア環境を対象としている．また，D-ARIES の論文は方式の提案にとどまっており，実装がなされておらず，性能評価が未記載である．

6.2 フラッシュストレージを前提とする研究

フラッシュストレージを対象とする研究には，文献 [1], [14], [15] がある．FlashLogging [1] は，複数のフラッシュストレージをそれぞれログデバイスに用いて，ログの書き込み対象をラウンドロビン方式で割り当てる．従来のディスクアレイ（RAID）の問題として，小さいサイズの書き込みでも複数ストライプにまたがることにより書き込みが分割される点や，ストライプをまたがない小さいサイズの書き込みリクエストが連続した場合に書き込みが 1 つのディスクに集中してしまう点をあげている．先行する書き込みが終わるまで，後に続く書き込みは結果を返すことができないという制約があるものの，書き込み完了の順番は前後することを許容し，並列に異なるデバイスへログを書き込むことで，書き込み処理をパイプライン化する．異なる領域に並列にログを書き込むことで性能を向上させる点は本研究と類似するが，本研究ではクライアントへの

コミット通知の順序決定をストレージ任せにせず，WAL機構内で制御することで，先行する書き込みの完了を待たずに，後に続く書き込みの結果を返すことができる．そのほか，FlashLoggingは単一のWALバッファを対象としており，WALバッファへのログレコード挿入の競合の緩和については考えられていない．

In-Page Logging [14], [15] はフラッシュメモリの特性に着目し，FTLを独自に設計している．フラッシュストレージのページや消去単位ごとにログ領域を持たせ，ストレージ上のデータの更新はログの追記として実現することで，フラッシュメモリの書き換えにともなう高コストな消去操作の回数を減らす．トランザクションのすべてのログがそれぞれのログ領域に書き込まれたことをコミット条件とすることで，トランザクション処理に対応する．ログを異なる領域に並列に書き込む点は本研究と類似するが，ログをページや消去単位ごとに書き込む必要があるので，非連続な小さいログ書き込みが多数発生する．そのため，スループット向上を目的としたグループコミットなどの複数ログの一括書き込みを行うことが難しい．

6.3 NVMを前提とする研究

バイト単位でアクセス可能なNVM（Non Volatile Memory）[13], [17], [24], [27] は実用化が近いと期待されており，NVMを用いてWALを高性能化する研究が近年活発に行われている [7], [9], [26]．Huangら [7] はログバッファをNVMに設置してWALを実行するNV-Loggingを提案している．CPUキャッシュにおけるwrite命令のリオーダにともなうデータ一貫性問題を回避するため，NV-Loggingはclflush命令を用いてログレコードの永続化を行うflush-on-insertとflush-on-commitを提供する．NVMでは永続化に要する時間が短いため，グループコミットやELRはNV-Loggingでは不要である旨が述べられている．この研究はNVMを前提としており，本研究で対象としているフラッシュストレージをはじめ既存のブロックデバイスに用いることができない．

Wangら [26] は複数のWALバッファをNVM内に用意し，その中へログレコードを並列に書き込むコミットプロトコルであるpassive group commitを提案している．Passive group commitはNVM上にWALバッファを置き，ログをDRAMを経由せずに直接NVMに書き込む．Passive group commitはELRであり，commit条件を満たしたかどうかをまとめて確認する専用のデーモンを用意する．本研究ではDRAMにWALバッファを設置する必要があるため，passive group commitをそのまま本研究の環境で用いることはできない．また，シーケンス番号としてページとトランザクションの両方で用いるGSNなるログ識別子を導入している．GSNは論理クロック [12] に基づいて計算され，その計算コストは単純なatomic fetch_and_addによるインクリメントよりも複雑で高コストである．

Kimura [9] はNVM向けの高速OLTPシステムであるFOEDUSを報告している．FOEDUSのコミットプロトコルはSilo [25], [28] を基礎としており，40 msの時区間（epoch）と楽観的並行性制御 [11] を組み合わせる．FOEDUSは同一epochのログレコードを一括してグループコミットする方式を採用している．Epoch番号の採番はログ単位およびトランザクション単位で採番するよりも頻度が小さいため，many-core環境においても採番によるスケーラビリティの阻害を避けることができる．この方式では同一epoch内におけるトランザクションの順序関係がログからは判断できないが，それらのトランザクションはepoch内のすべてのログを永続化してはじめてcommitしたと見なすことで，その判断を不要にしている．その代償として，どれほど短いトランザクションでも，応答遅延はAether [8] に比べて大きくなる．なぜならepochが完了するまで，クライアントへの応答通知が待たされるからである．本研究はepoch方式と比較して，遅延をできるだけ犠牲にせずに，スループットを向上させる手法である．

7. 結論

本研究ではマルチコア共有マシン上で，ブロック単位でアクセスするストレージデバイス，特に性能を引き出すことができるフラッシュストレージを対象に，WALの並列化手法P-WALを提案した．

P-WALはフラッシュストレージの特性を活用し，各ワーカが専用の領域にログを書き込む並列ログ書き込み方式を用いる．この方式により従来の直列WAL方式で発生する，排他制御処理とストレージI/Oにともなう性能低下問題を解決した．

P-WALの性能をプロトタイプのトランザクションマネージャ上で評価した．マイクロベンチマークU-1において，P-WALは最大1.82 M tpsを示した．これは直列WAL方式Aetherに対して10.0倍の性能向上であった．TPC-Cベンチマークにおいて，P-WALは最大67.8 K tpsを示した．これはAetherに対して2.3倍の性能向上であった．

P-WALは遅延の増加を比較的避けながらスループットを向上させる技法に位置づけられる．我々はP-WALがARIES方式を採用している既存トランザクションシステムの高性能化に貢献できると考える．

謝辞　本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」，JST CREST「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」，科研費「#16K00150」によるものである．また，丁寧な査読と有益なコメントをくださった査読者に感謝する．

参考文献

[1] Chen, S.: FlashLogging — exploiting flash devices for synchronous logging performance, *SIGMOD Conference* (2009).

[2] Dell EMC: Rack-Scale Flash Technology — DSSD D5, available from ⟨http://www.emc.com/en-us/storage/flash/dssd/dssd-d5/technology.htm⟩ (accessed 2016-12-02).

[3] Fusion-io: Application Acceleration — Enterprise Flash Memory Platform — Fusion-io, available from ⟨http://www.fusionio.com/⟩ (accessed 2015-04-15).

[4] Gray, J.: The Transaction Concept: Virtues and Limitations, *VLDB*, pp.144–154 (1981).

[5] Hagmann, R.: Reimplementing the cedar file system using logging and group commit, *SOSP*, pp.155–162 (1987).

[6] Harizopoulos, S., Abadi, D.J., Madden, S. and Stonebraker, M.: OLTP through the looking glass, and what we found there, *SIGMOD Conference* (2008).

[7] Huang, J., Schwan, K. and Qureshi, M.K.: NVRAM-aware Logging in Transaction Systems, *VLDB Endowment*, Vol.8, No.4, pp.389–400 (2014).

[8] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M. and Ailamaki, A.: Aether: a scalable approach to logging, *VLDB*, Vol.3, pp.681–692 (2010).

[9] Kimura, H.: Foedus: OLTP engine for a thousand cores and nvram, *SIGMOD Conference*, pp.691–706 (2015).

[10] Koopman, P.: 32-Bit Cyclic Redundancy Codes for Internet Applications, *DSN* (2002).

[11] Kung, H.T. and Robinson, J.T.: On Optimistic Methods for Concurrency Control, *ACM Trans. Database Syst.*, Vol.6, No.2, pp.213–226 (1981).

[12] Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).

[13] Lee, B.C., Ipek, E., Mutlu, O. and Burger, D.: Architecting phase change memory as a scalable dram alternative, *ISCA* (2009).

[14] Lee, S.-W. and Moon, B.: Design of Flash-based DBMS: An In-page Logging Approach, *SIGMOD Conference*, pp.55–66 (2007).

[15] Lee, S.-W. and Moon, B.: Transactional In-Page Logging for Multiversion Read Consistency and Recovery, *ICDE*, pp.876–887 (2011).

[16] MariaDB: Fusion-io NVMFS Atomic Write Support — MariaDB Knowledge Base, available from ⟨https://mariadb.com/kb/en/mariadb/fusion-io-nvmfs-atomic-write-support/⟩ (accessed 2016-12-02).

[17] Mishra, A.K., Dong, X., Sun, G., Xie, Y., Vijaykrishnan, N. and Das, C.R.: Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs, *ACM SIGARCH Computer Architecture News*, Vol.39, No.3, pp.69–80 (2011).

[18] Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H. and Schwarz, P.M.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, *ACM TODS*, Vol.17, No.1, pp.94–162 (1992).

[19] NVM Express: NVM Express — scalable, efficient, and industry standard, available from ⟨http://www.nvmexpress.org/⟩ (accessed 2016-12-02).

[20] OpenNVM: OpenNVM Project, available from ⟨http://opennvm.github.io⟩ (accessed 2016-06-16).

[21] Rafii, A. and DuBois, D.: Performance tradeoffs of group commit logging, *CMG Conference* (1989).

[22] Sancho, J.C. and Kerbyson, D.J.: Analysis of double buffering on two different multicore architectures — Quad-core Opteron and the Cell-BE, *IPDPS*, pp.1–12 (2008).

[23] Speer, J. and Kirchberg, M.: D-ARIES - A Distributed Version of the ARIES Recovery Algorithm, *ADBIS Research Communications* (2005).

[24] Strukov, D.B., Snider, G.S., Stewart, D.R. and Williams, R.S.: The missing memristor found, *Nature*, Vol.453, No.7191, pp.80–83 (2008).

[25] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy transactions in multicore in-memory databases, *SOSP*, pp.18–32 (2013).

[26] Wang, T. and Johnson, R.: Scalable logging through emerging non-volatile memory, *VLDB Endowment*, Vol.7, No.10, pp.865–876 (2014).

[27] Wong, H.S.P., Raoux, S., Kim, S., Liang, J., Reifenberg, J.P., Rajendran, B., Asheghi, M. and Goodson, K.E.: Phase Change Memory, *Proc. IEEE*, Vol.98, No.12, pp.2201–2227 (2010).

[28] Zheng, W., Tu, S., Kohler, E. and Liskov, B.: Fast databases with fast durability and recovery through multicore parallelism, *OSDI*, pp.465–477 (2014).

神谷 孝明　（学生会員）

1993 年生．2015 年筑波大学情報学群情報科学類卒業．同年同大学大学院システム情報工学研究科コンピュータサイエンス専攻入学，現在に至る．

川島 英之　（正会員）

1999 年慶應義塾大学理工学部電気工学科卒業．2005 年同大学大学院理工学研究科開放環境科学専攻後期博士課程修了．同年慶應義塾大学理工学部助手．2007 年筑波大学大学院システム情報工学研究講師，ならびに計算科学研究センター講師．2016 年筑波大学計算科学研究センター准教授．博士（工学）．データ基盤に興味を持つ．電子情報通信学会，ACM，IEEE 各会員．

星野 喬

1981 年生．2003 年東京大学工学部電子情報工学科卒業．2005 年同大学大学院修士課程修了．2008 年同大学院博士課程単位取得後退学．2009 年サイボウズ・ラボ入社．主にソフトウェア開発に従事．ACM 会員．

建部 修見　（正会員）

1969 年生．1992 年東京大学理学部情報科学科卒業．1997 年同大学大学院理学系研究科情報科学専攻博士課程修了．同年電子技術総合研究所入所．2005 年独立行政法人産業技術総合研究所主任研究員．2006 年筑波大学大学院システム情報工学研究科准教授．2015 年同教授．博士（理学）（東京大学）．超高速計算システム，グリッドコンピューティング，並列分散システムソフトウェアの研究に従事．日本応用数理学会，ACM 各会員．

（担当編集委員　藤原 真二）